Classification

(supervised learning)

JJ Valletta

March 4, 2015

www.exeter.ac.uk/as/rdp/

Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

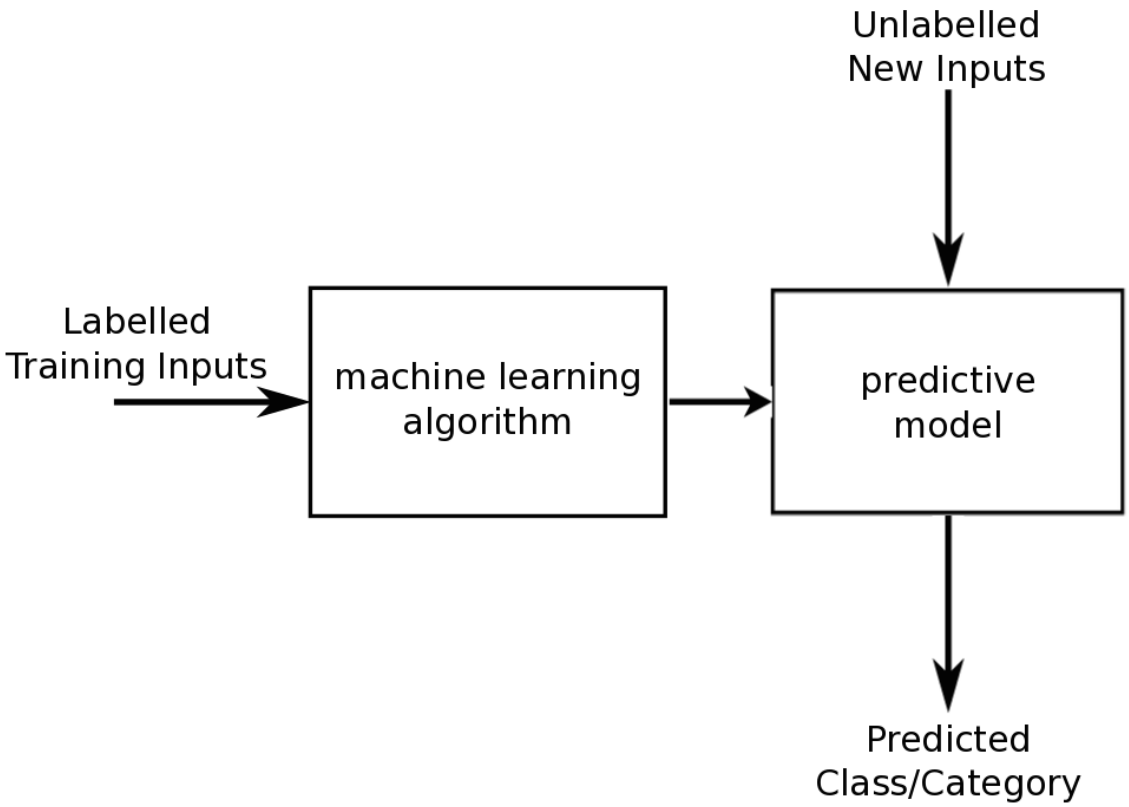
Overview

- The classification task
- Over/under-fitting a predictive model
- k-nearest neighbour (kNN)
- Decision trees
- Random forests
- Support vector machines (SVM)

Classification

Definition

Given a set of observations assign each one to a predefined category/class. A classifier/predictive model built using labelled inputs does the job.



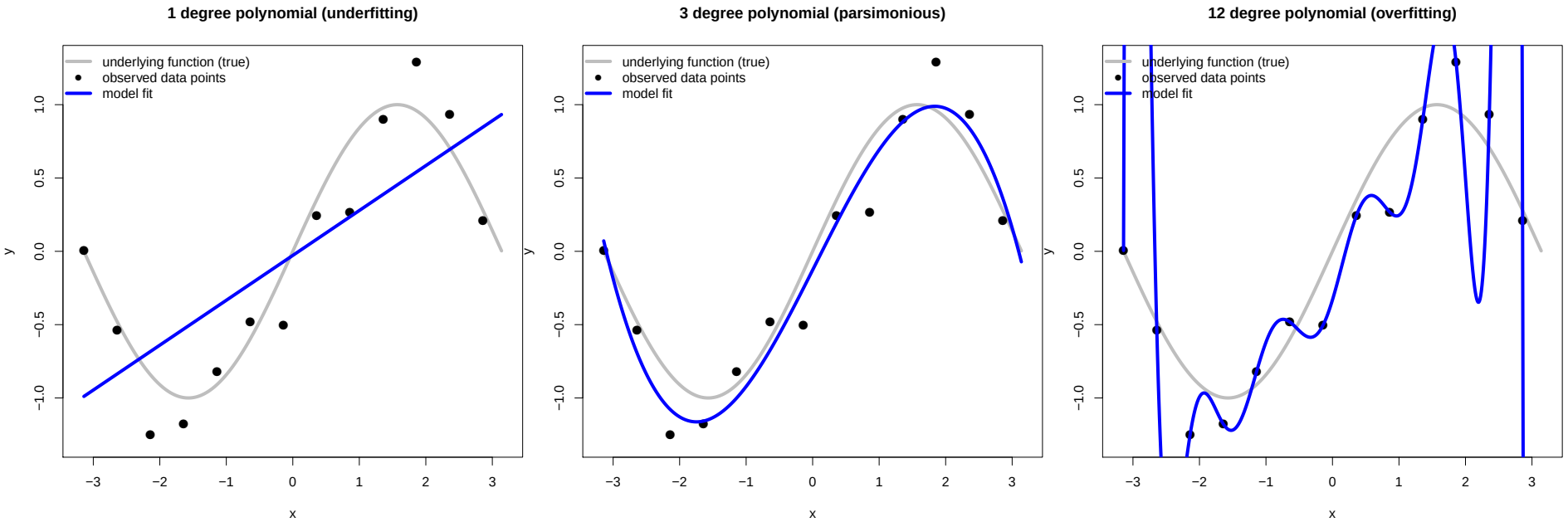
Note: Better data/features will almost always beat better algorithms

Where are classifiers used?

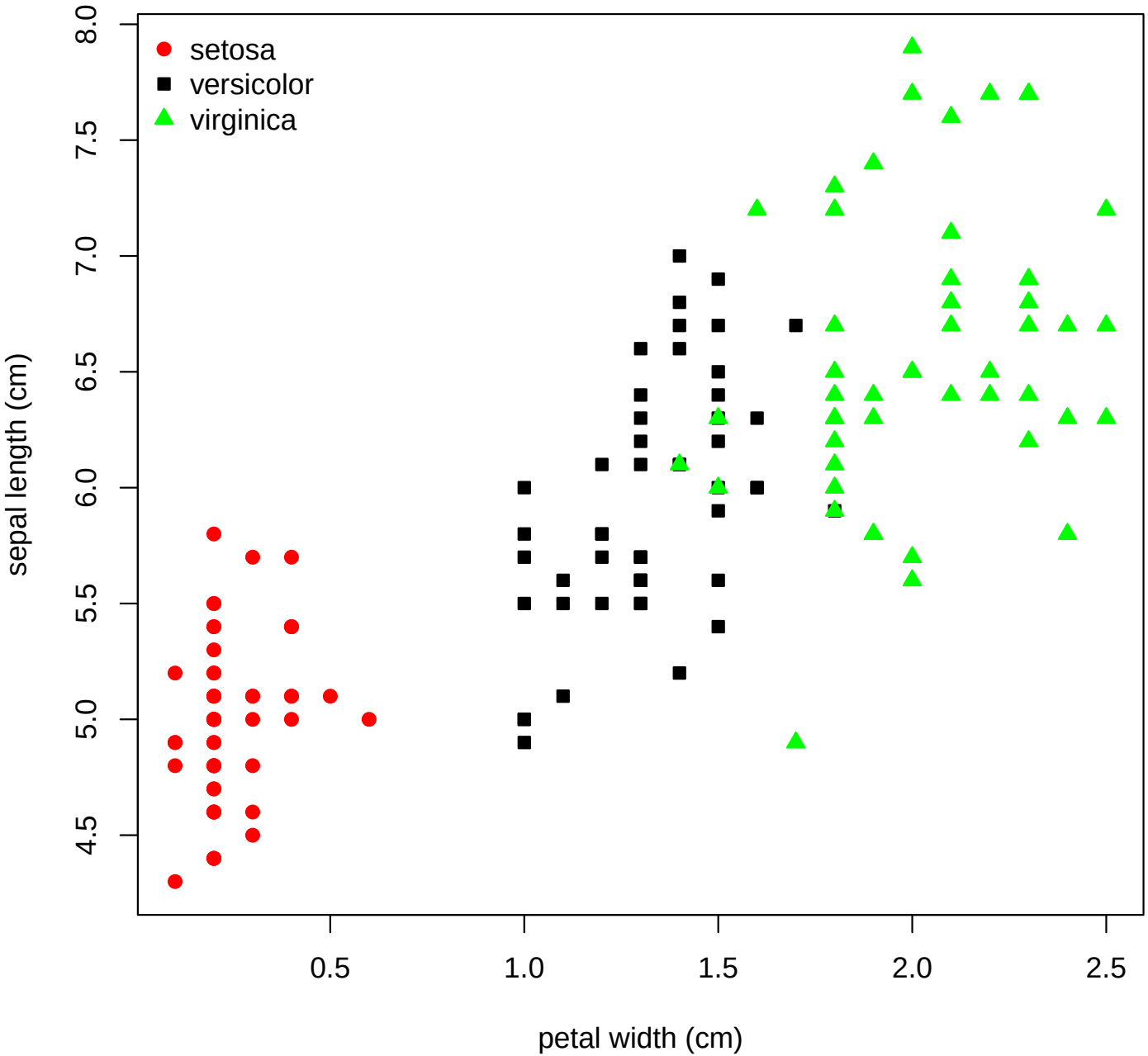
Medical imaging:	is tumour benign or cancerous
Gene expression:	use "signature" to classify patient as having (or not) a condition
Computer vision:	detect and track a moving object
Biogeography:	classify land cover using remote sensing imagery
Speech recognition:	translate audio signals into written text
Biometric authentication:	identify a person using some personal characteristic e.g fingerprint or DNA
Epidemiology:	given a set of risk factors what is the chance of patient suffering from a certain condition

Over/under-fitting (bias-variance tradeoff)

- How well should we fit the training data to get good generalisation?
- Driving training error to zero is not a good idea
- **Bias** caused by a too rigid model leads to *underfitting*
- **Variance** caused by a too flexible model leads to *overfitting*
- **Occam's Razor**, pick the simplest model that explains your data, *parsimony*



Iris dataset



k-nearest neighbour (kNN)

```
library(class)
fit <- knn(train, test, cl, k)
# train - training dataset (matrix or data frame)
# test - testing dataset (matrix or data frame)
# cl - corresponding label of the training dataset (factor)
# k - number of neighbours
```

1. Calculate distance between test point and every training data point
2. Find the k training points closest to test point
3. Assign test point the majority vote of their class label

k-nearest neighbour (kNN)

```
library(class)
fit <- knn(train, test, cl, k)
# train - training dataset (matrix or data frame)
# test - testing dataset (matrix or data frame)
# cl - corresponding label of the training dataset (factor)
# k - number of neighbours
```

1. Calculate distance between test point and every training data point
2. Find the k training points closest to test point
3. Assign test point the majority vote of their class label

k-nearest neighbour (kNN)

```
library(class)
fit <- knn(train, test, cl, k)
# train - training dataset (matrix or data frame)
# test - testing dataset (matrix or data frame)
# cl - corresponding label of the training dataset (factor)
# k - number of neighbours
```

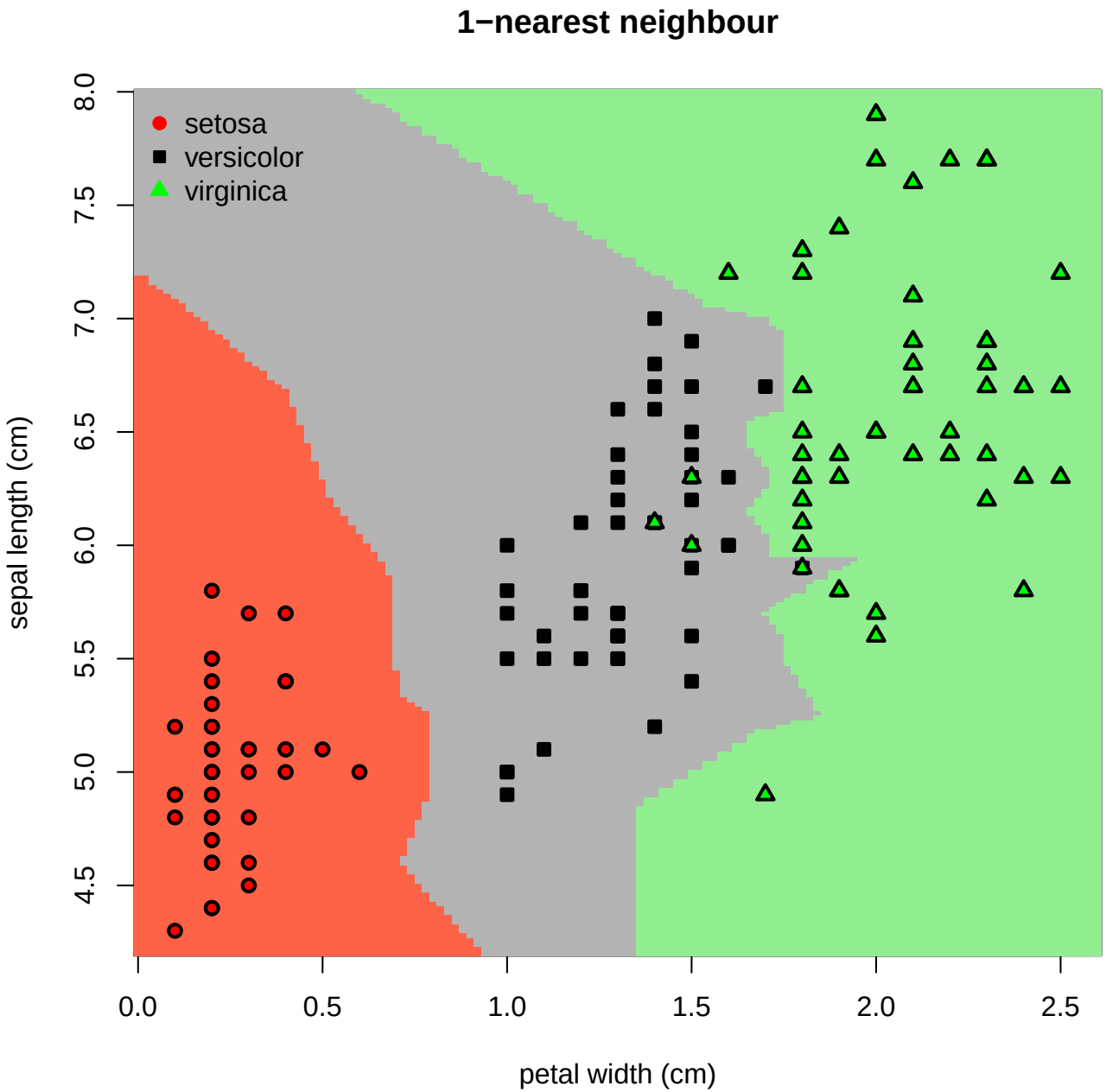
1. Calculate distance between test point and every training data point
2. Find the k training points closest to test point
3. Assign test point the majority vote of their class label

k-nearest neighbour (kNN)

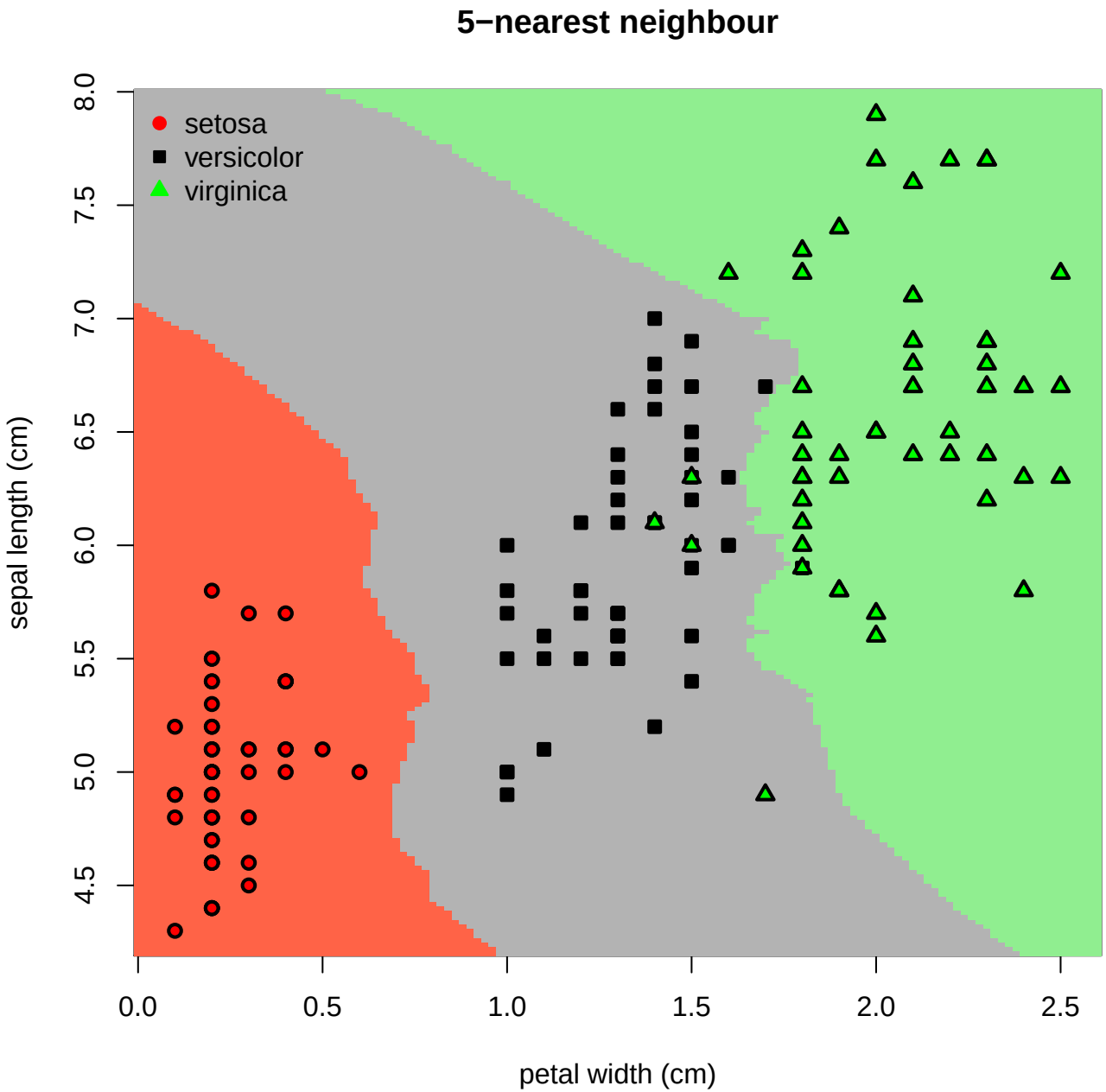
```
library(class)
fit <- knn(train, test, cl, k)
# train - training dataset (matrix or data frame)
# test - testing dataset (matrix or data frame)
# cl - corresponding label of the training dataset (factor)
# k - number of neighbours
```

1. Calculate distance between test point and every training data point
2. Find the k training points closest to test point
3. Assign test point the majority vote of their class label

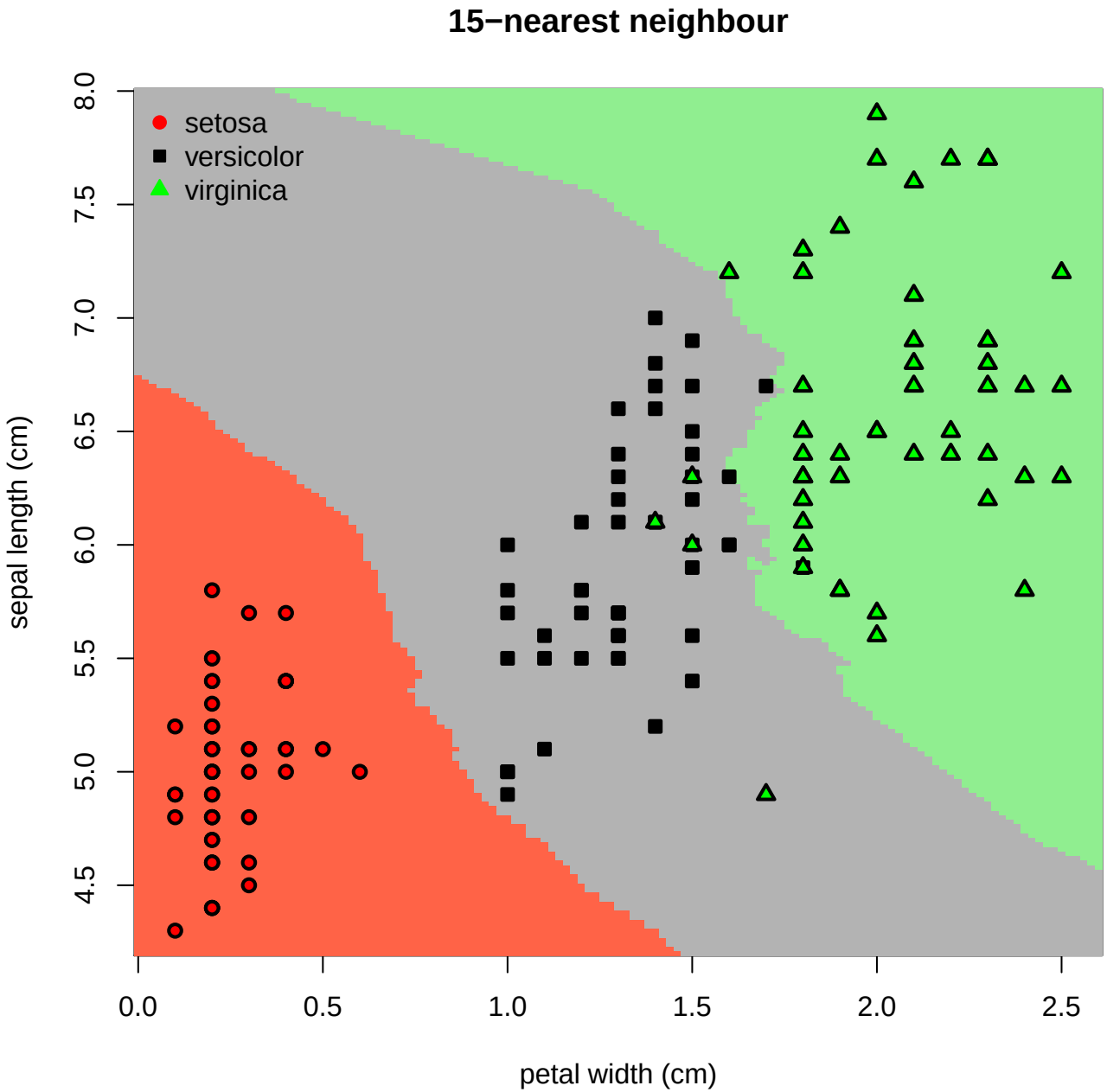
k-nearest neighbour



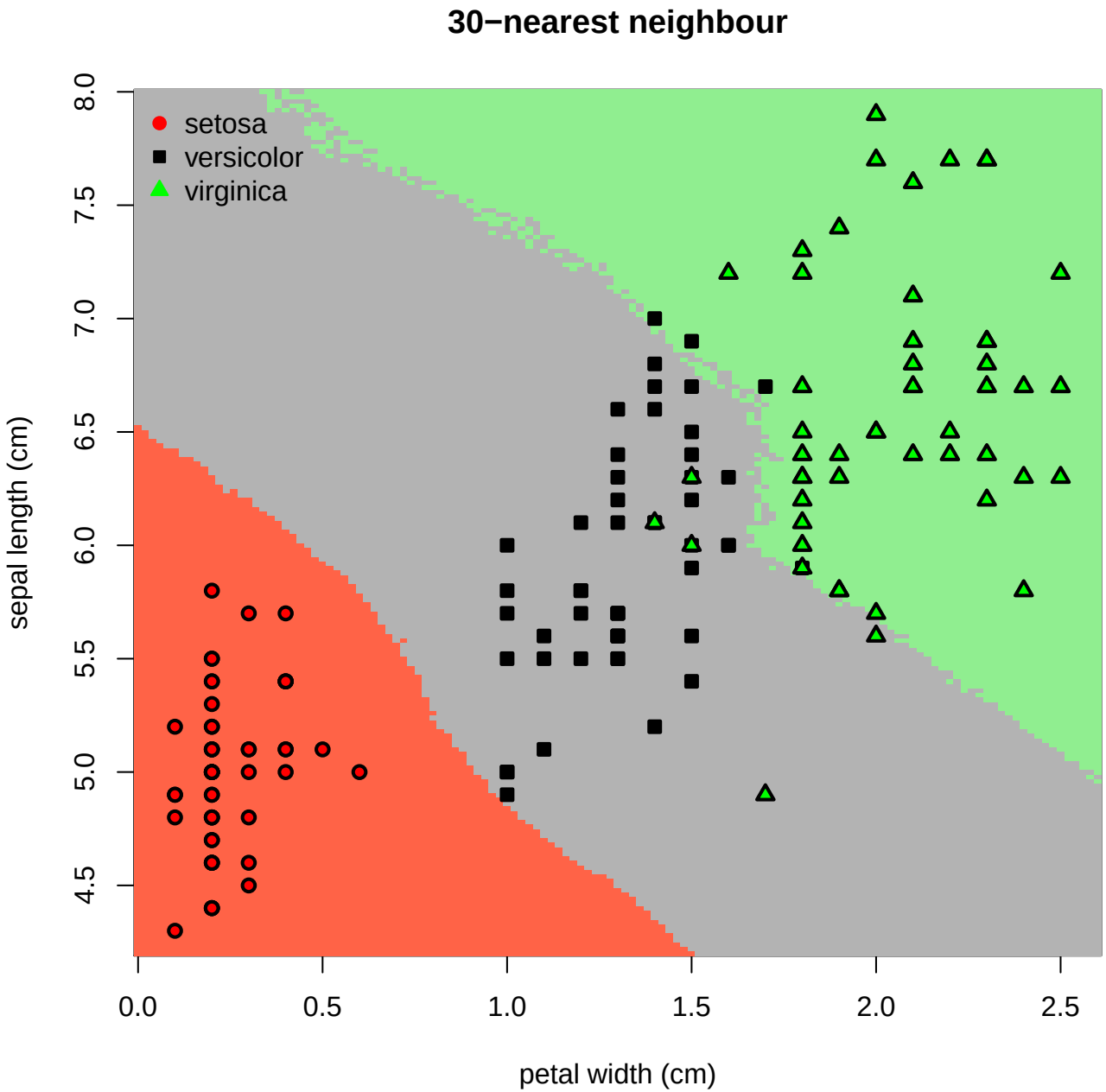
k-nearest neighbour



k-nearest neighbour



k-nearest neighbour



k-nearest neighbour

Pros

- Simple and intuitive
- Works for multi-class problems
- Non-linear decision boundaries
- k easily tuned by cross-validation

Cons

- Can be computationally expensive as for every test point distance to every training data points needs to be computed (the model is actually the whole training dataset)
- Defining nearest by a distance metric can be ambiguous (for e.g when you have categorical predictors)

Decision trees

```
library(tree)
fit <- tree(formula, data)
# OR
library(rpart)
fit <- rpart(formula, data)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
```

1. Divide data into left-right (yes-no) by an axis parallel split using *one* predictor
2. The best split is found by maximising information gain/lowering entropy
3. Repeat 1 to 2 until all data is correctly classified or some stopping rule reached

Decision trees

```
library(tree)
fit <- tree(formula, data)
# OR
library(rpart)
fit <- rpart(formula, data)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
```

1. Divide data into left-right (yes-no) by an axis parallel split using *one* predictor
2. The best split is found by maximising information gain/lowering entropy
3. Repeat 1 to 2 until all data is correctly classified or some stopping rule reached

Decision trees

```
library(tree)
fit <- tree(formula, data)
# OR
library(rpart)
fit <- rpart(formula, data)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
```

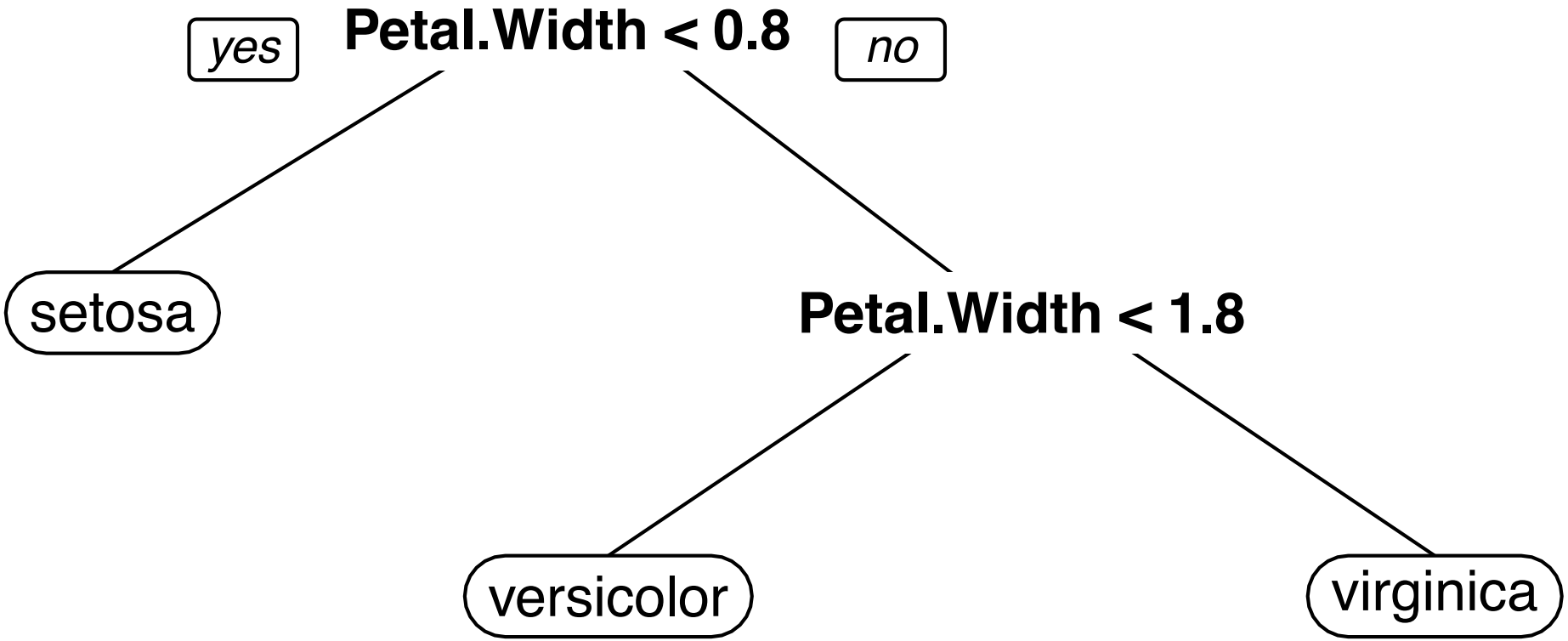
1. Divide data into left-right (yes-no) by an axis parallel split using *one* predictor
2. The best split is found by maximising information gain/lowering entropy
3. Repeat 1 to 2 until all data is correctly classified or some stopping rule reached

Decision trees

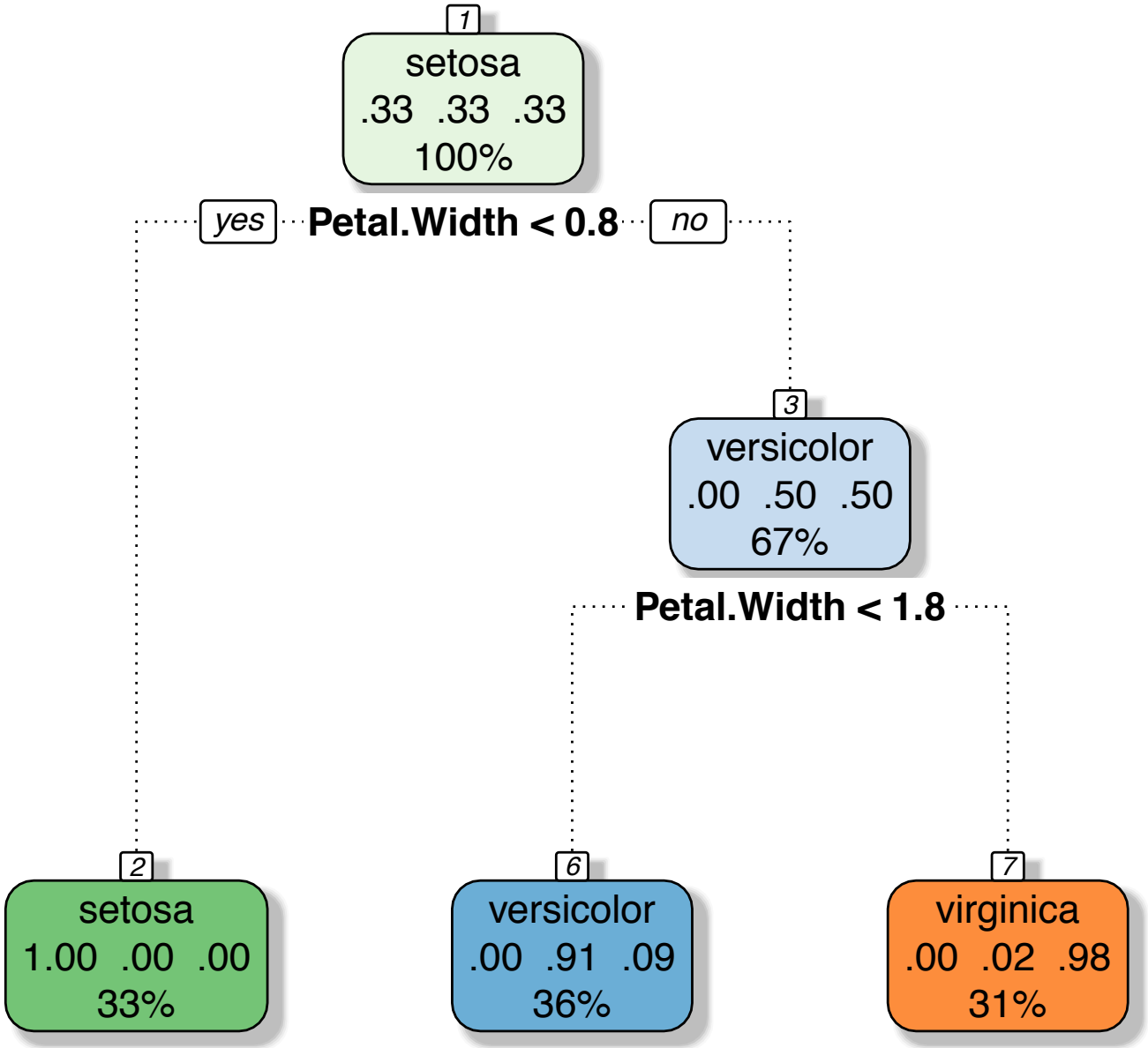
```
library(tree)
fit <- tree(formula, data)
# OR
library(rpart)
fit <- rpart(formula, data)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
```

1. Divide data into left-right (yes-no) by an axis parallel split using *one* predictor
2. The best split is found by maximising information gain/lowering entropy
3. Repeat 1 to 2 until all data is correctly classified or some stopping rule reached

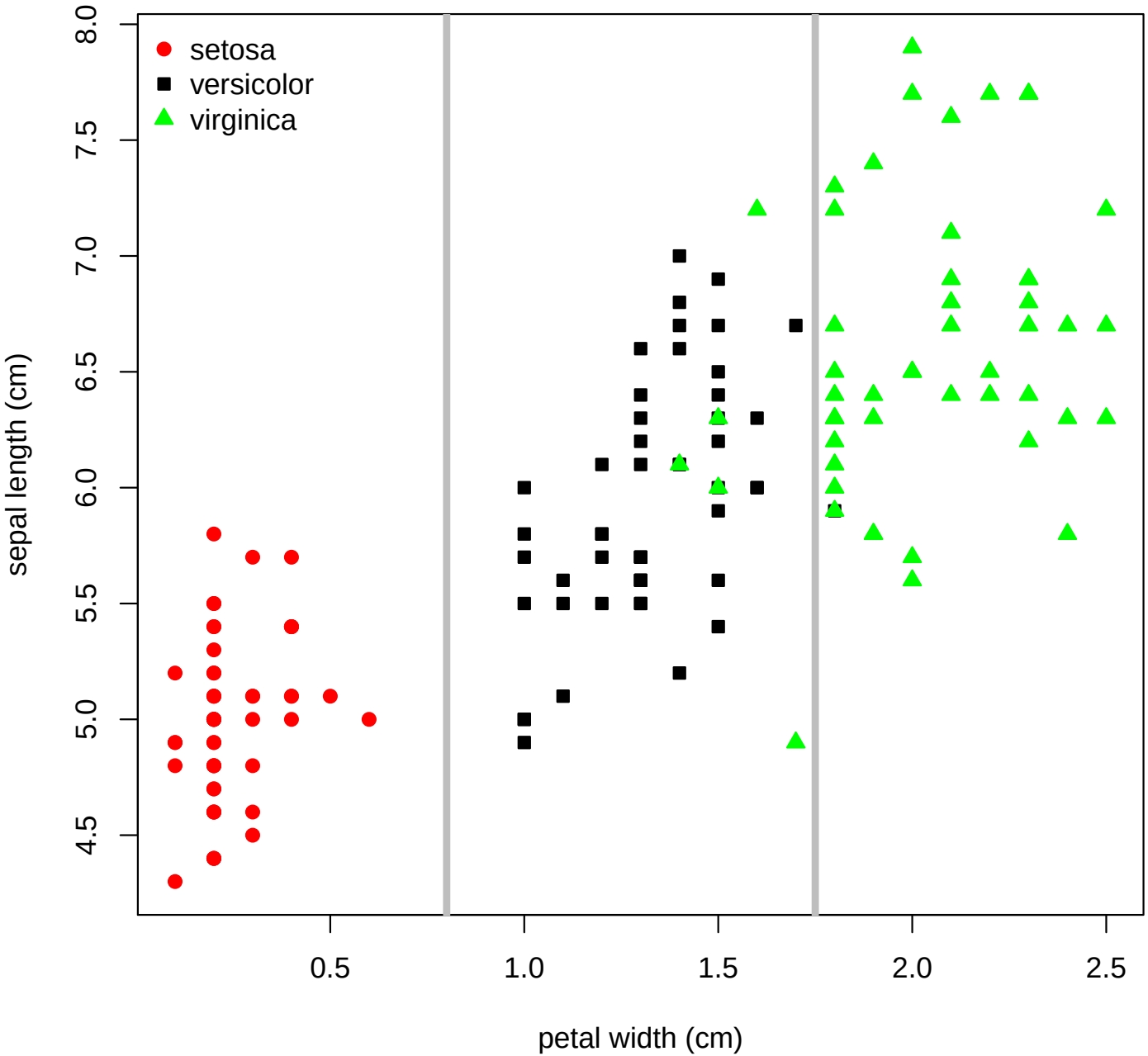
Decision trees - The Model



Decision trees - The Model



Decision boundaries



Decision trees

Pros

- Model is very interpretable and can be directly used to generate rules
- Computationaly inexpensive to train and evaluate
- Handle both categorical and continuous data
- Robust to outliers

Cons

- Can easily overfit the data
- Predictive accuracy can be poor
- Small changes to training data may lead to a completely different tree

Random forests

- Decision trees are intuitive but suffer from overfitting which significantly affect their predictive accuracy
- *Pruning*, to "trim" the tree back, help reduce this overfit
- *Ensemble* methods such as Random Forests are a better alternative
- **Rationale**: Instead of one tree, grow a *forest*, where every bushy tree (no pruning) is a bit different, then average predictions over all trees

$\Rightarrow$

Random forests

```
library(randomForest)
fit <- randomForest(formula, data, ntree, mtry)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
# ntree - number of trees in forest
# mtry - number of predictors randomly sampled as candidates at
    each split (default is sqrt(num of covariates))
```

1. Grow $ntree$ bushy trees (no pruning) which are de-correlated from each other
2. Forest randomness is induced by:
   - Bagging (Bootstrap AGGregatING): each tree is trained on a subset of the data randomly sampled with replacement
   - For every tree split, consider only $mtry$ predictors as candidates at tree split
3. Average predictions from all $ntree$ trees

Random forests

```
library(randomForest)
fit <- randomForest(formula, data, ntree, mtry)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
# ntree - number of trees in forest
# mtry - number of predictors randomly sampled as candidates at
    each split (default is sqrt(num of covariates))
```

1. Grow $ntree$ bushy trees (no pruning) which are de-correlated from each other
2. Forest randomness is induced by:
   - Bagging (Bootstrap AGGregatING): each tree is trained on a subset of the data randomly sampled with replacement
   - For every tree split, consider only $mtry$ predictors, as candidates for the split
3. Average predictions from all $ntree$ trees

Random forests

```
library(randomForest)
fit <- randomForest(formula, data, ntree, mtry)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
# ntree - number of trees in forest
# mtry - number of predictors randomly sampled as candidates at
    each split (default is sqrt(num of covariates))
```

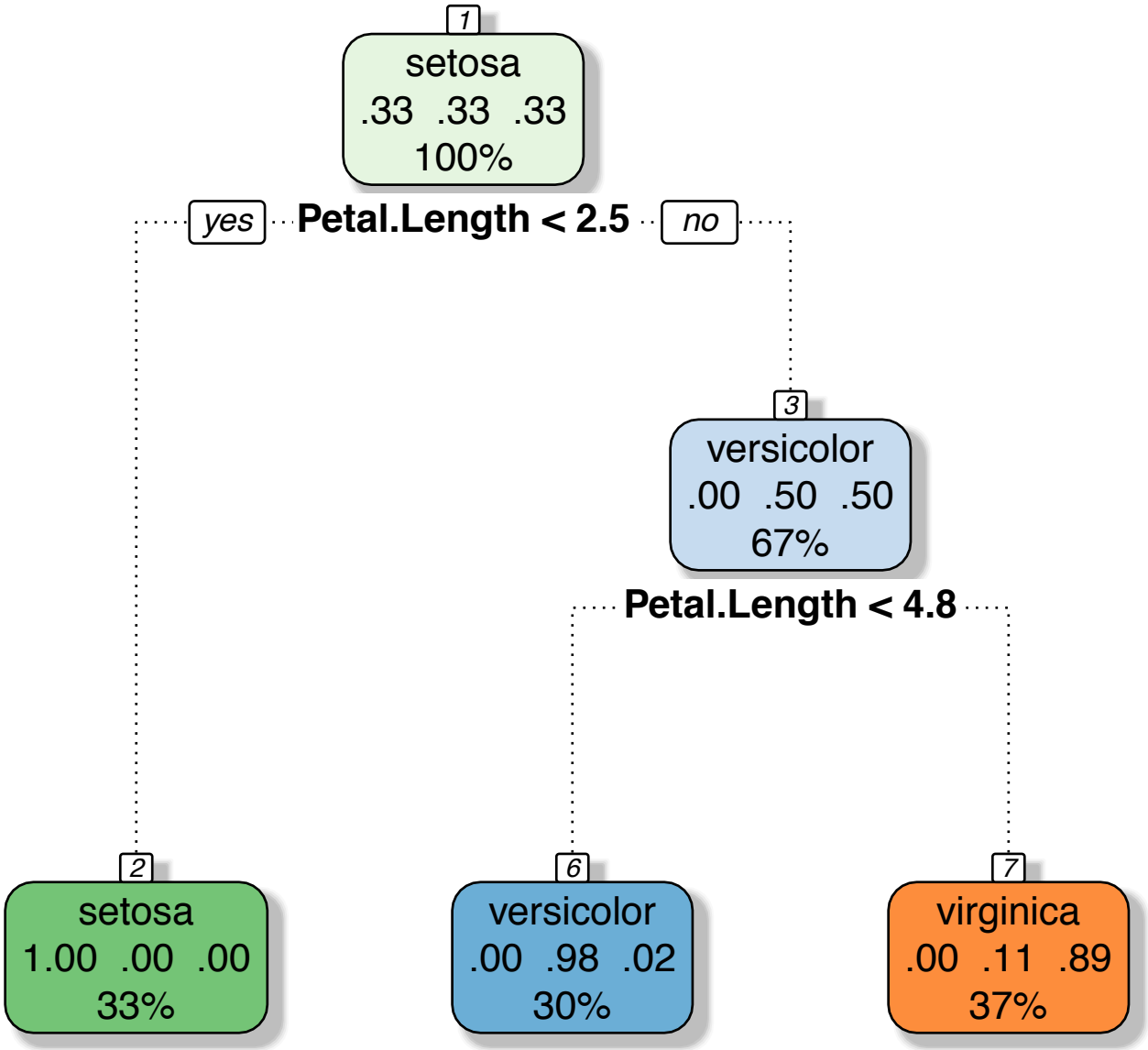
1. Grow $ntree$ bushy trees (no pruning) which are de-correlated from each other
2. Forest randomness is induced by:
   - Bagging (**B**ootstrap **AGG**regat**ING**), each tree is trained on a subset of the data randomly sampled with replacement
   - For every tree split consider only $mtry$ predictors as candidates for that split
3. Average predictions from all $ntree$ trees

Random forests

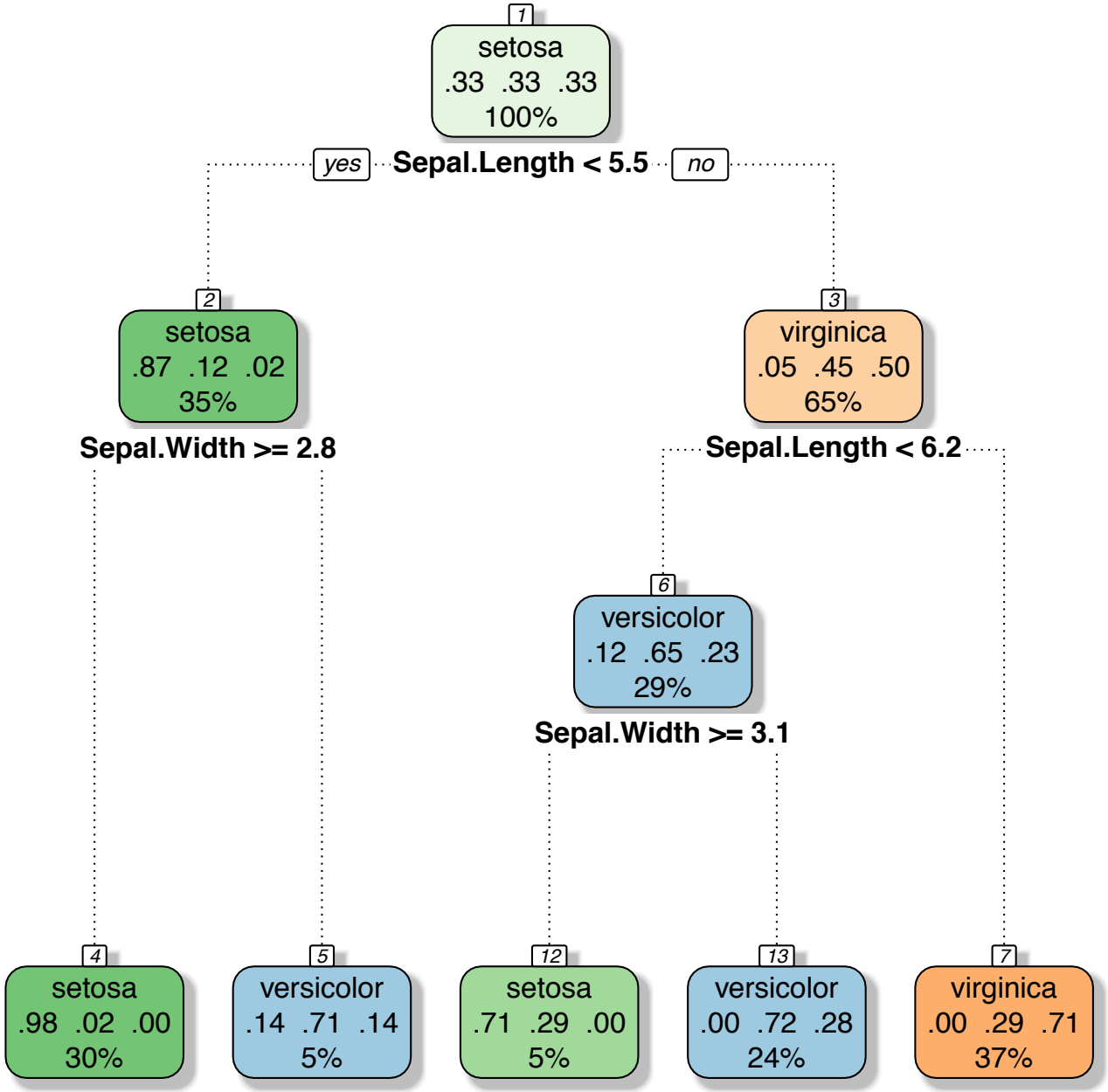
```
library(randomForest)
fit <- randomForest(formula, data, ntree, mtry)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
# ntree - number of trees in forest
# mtry - number of predictors randomly sampled as candidates at
    each split (default is sqrt(num of covariates))
```

1. Grow $ntree$ bushy trees (no pruning) which are de-correlated from each other
2. Forest randomness is induced by:
   - Bagging (**B**ootstrap **AGG**regat**ING**), each tree is trained on a subset of the data randomly sampled with replacement
   - For every tree split consider only $mtry$ predictors as candidates for that split
3. Average predictions from all $ntree$ trees

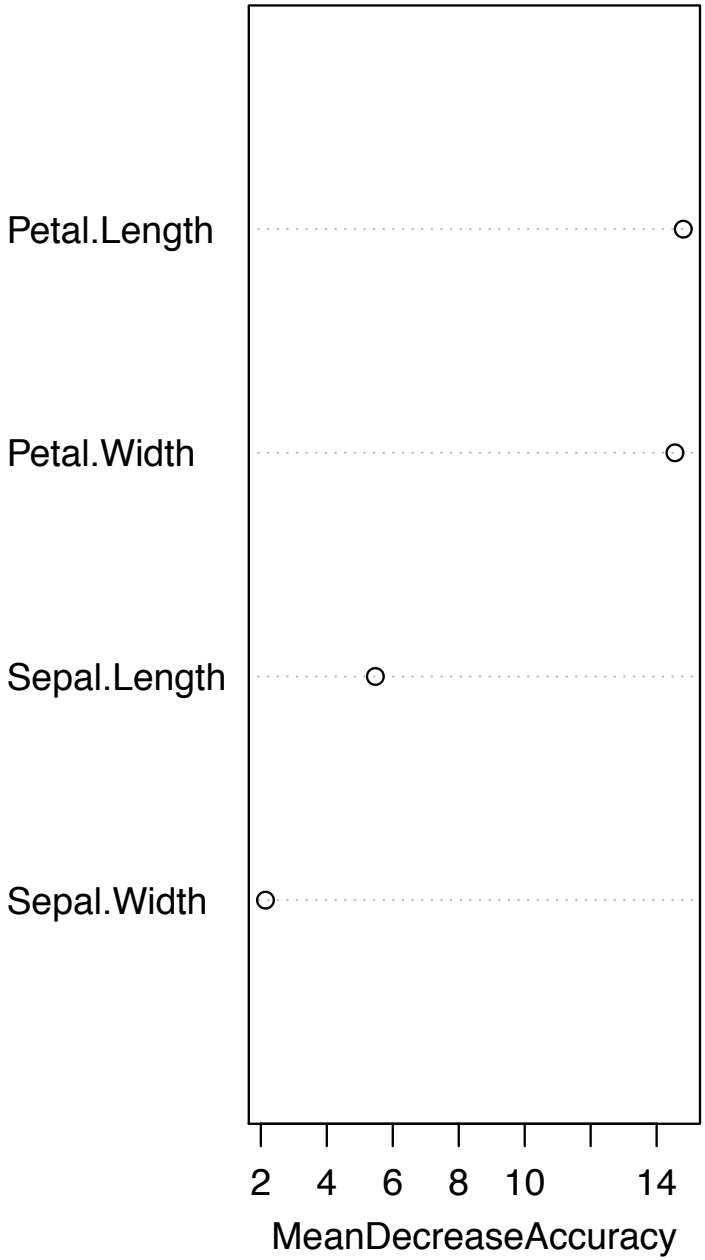
De-correlated bushy trees in the forest

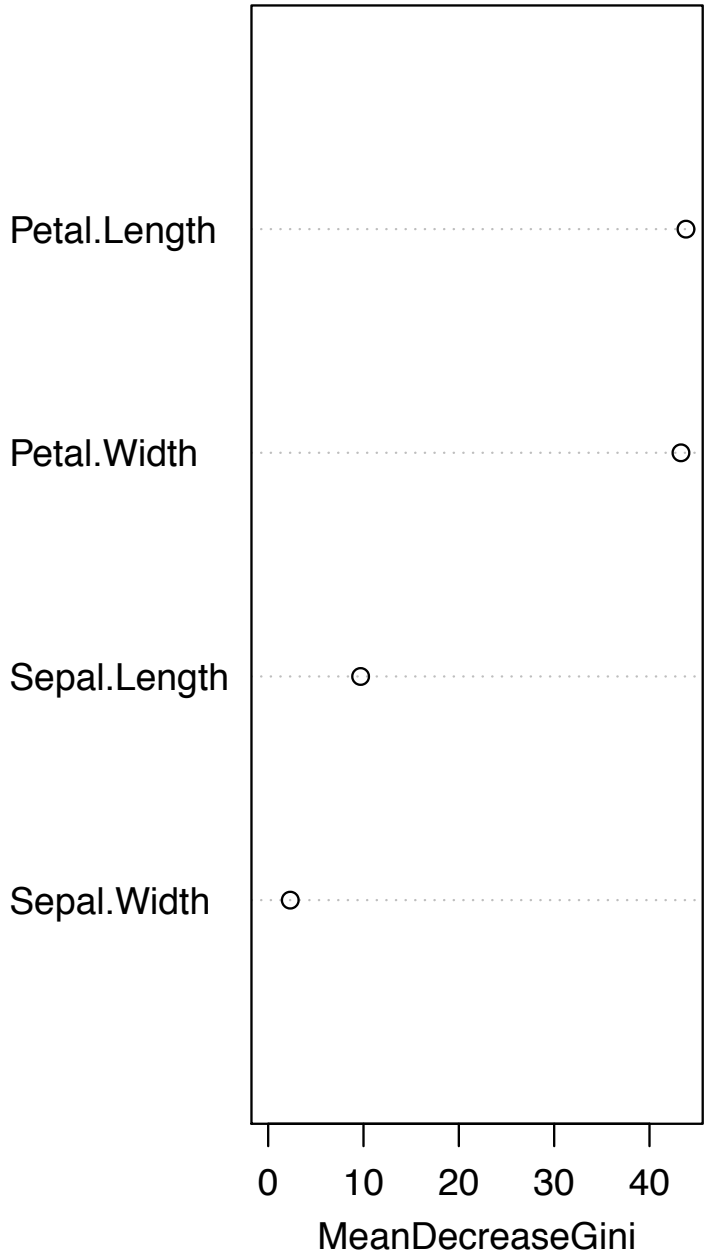


De-correlated bushy trees in the forest



Variable importance





Random forests

Pros

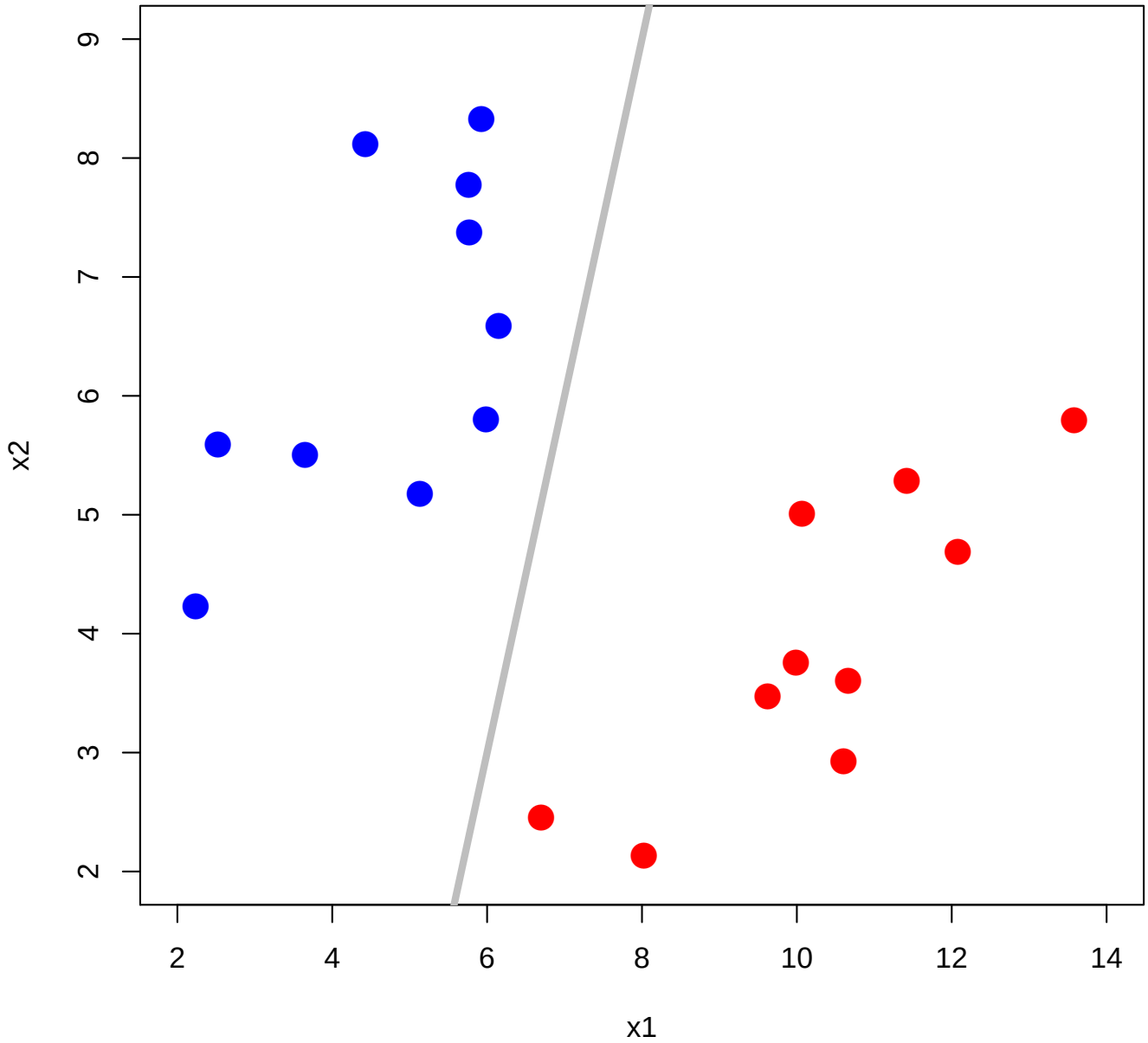
- State-of-the-art predictive accuracy
- Can handle thousands of both categorical and continuous predictors without variable deletion
- Robust to outliers
- Estimates the importance of every predictor
- Out-of-bag error (unbiased estimate of test error for every tree built)
- Can cope with unbalanced datasets by setting class weights

Cons

- Harder to interpret then plain decision trees

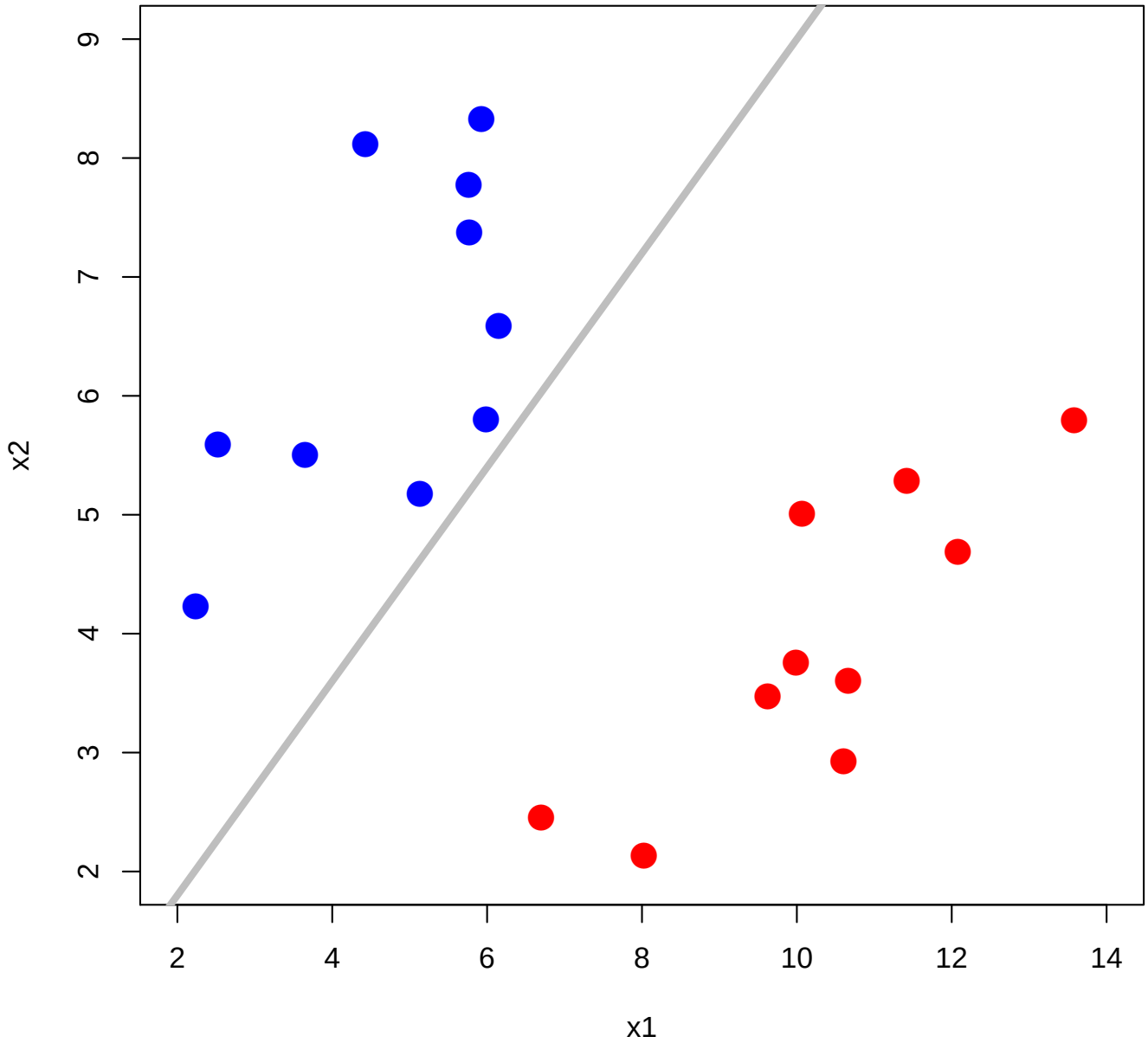
Support vector machines (SVM)

Which is the best separating line?



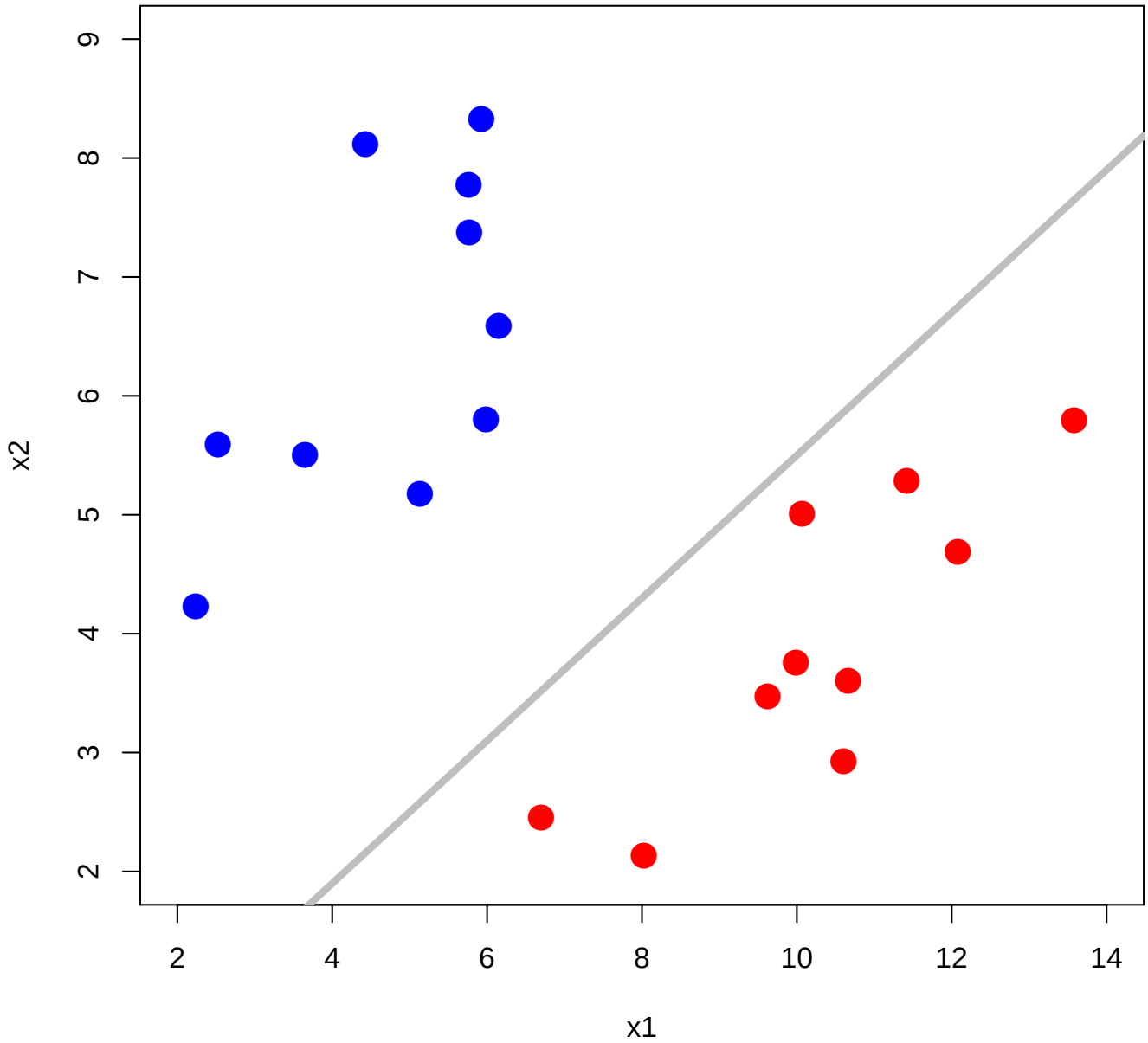
Support vector machines (SVM)

Which is the best separating line?



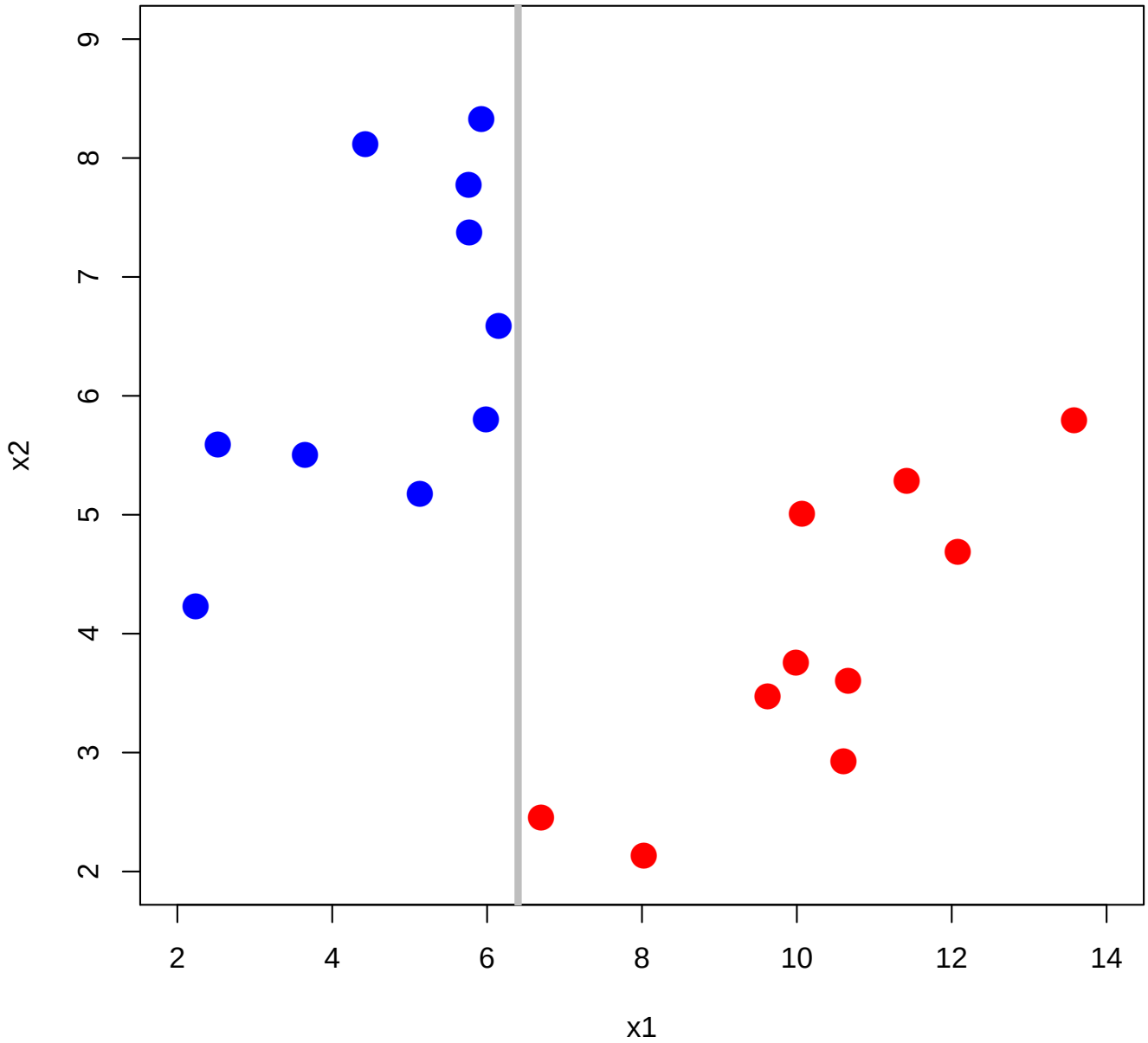
Support vector machines (SVM)

Which is the best separating line?



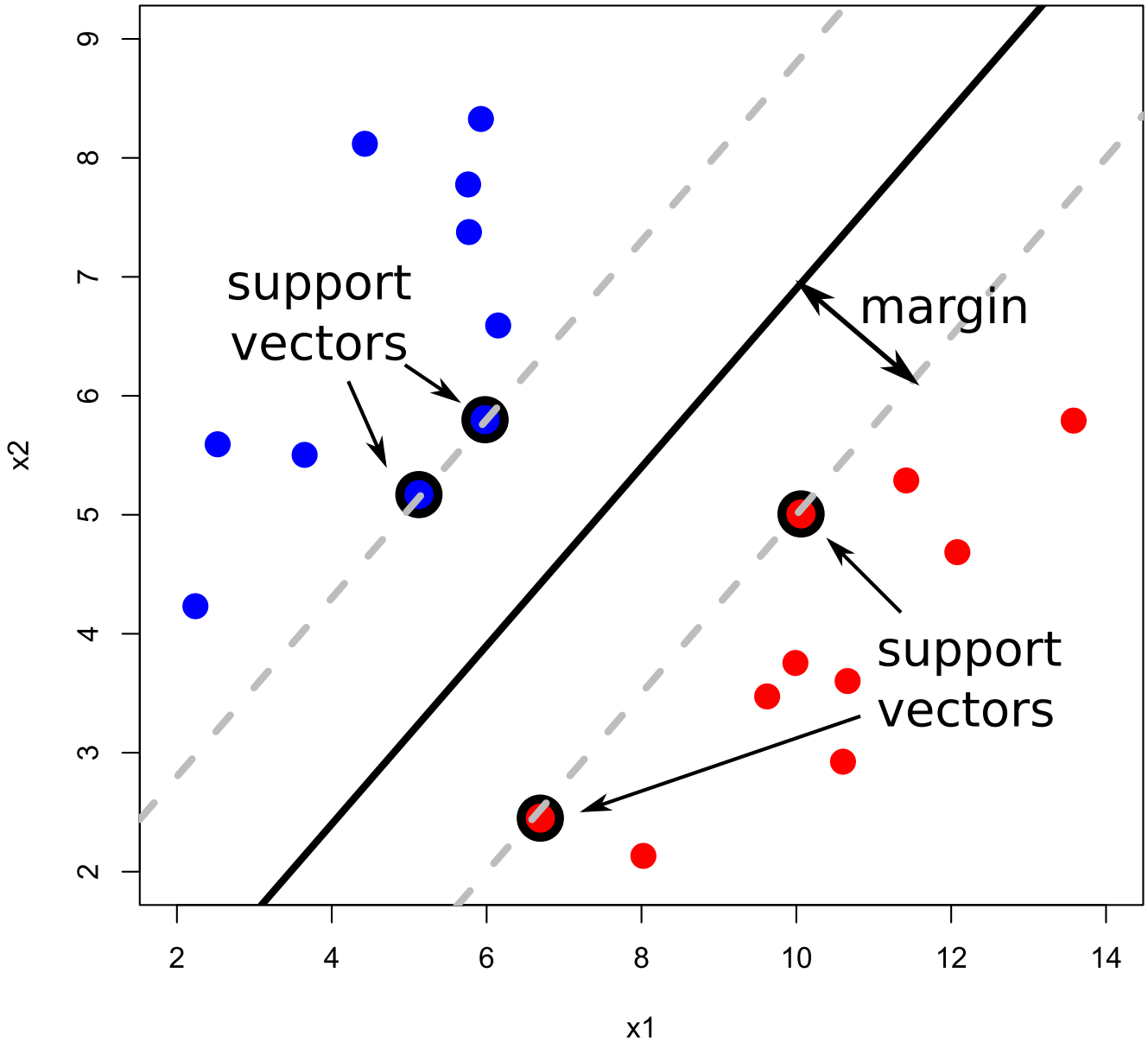
Support vector machines (SVM)

Which is the best separating line?



Support vector machines

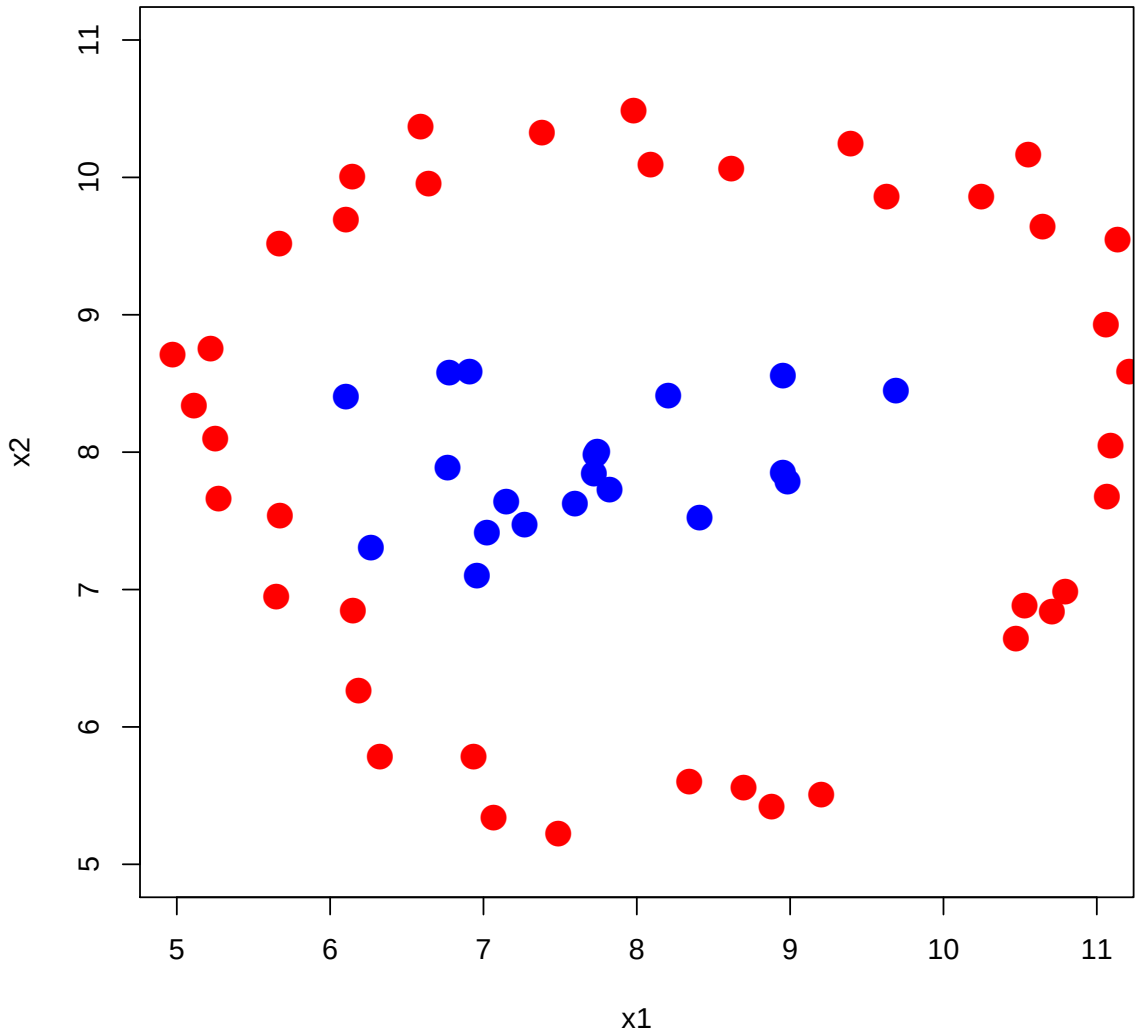
Rationale: Maximise the *margin*, the distance to separating hyperplane



Support vector machines

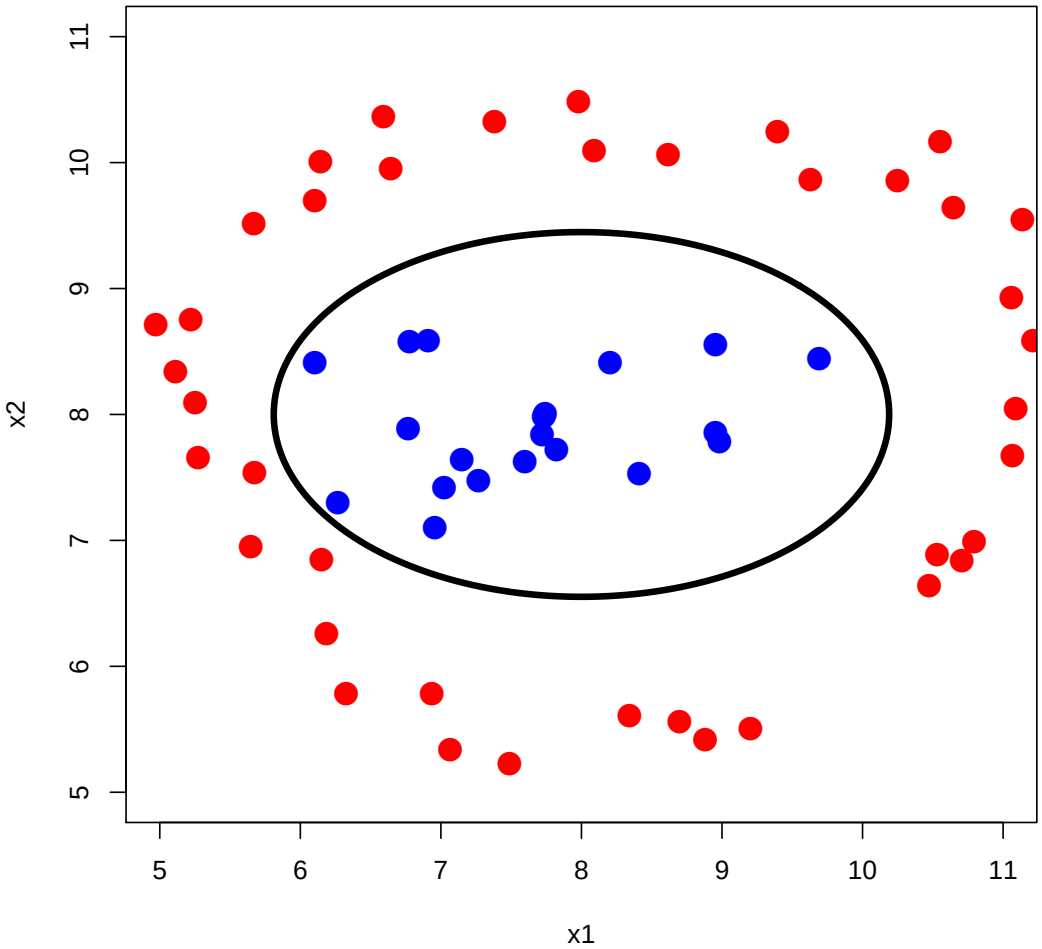
If it's something weird and it don't look good. Who ya gonna call? Ghostbusters?

Close, we need alternate dimensions to make them linearly separable

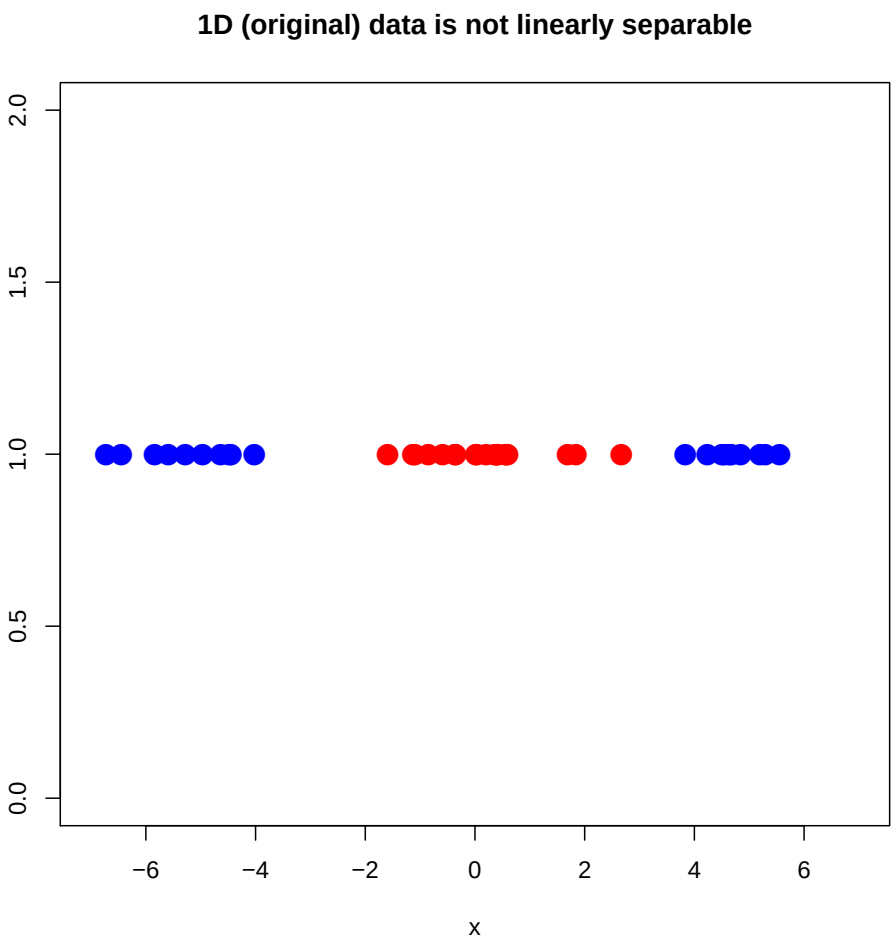


Support vector machines

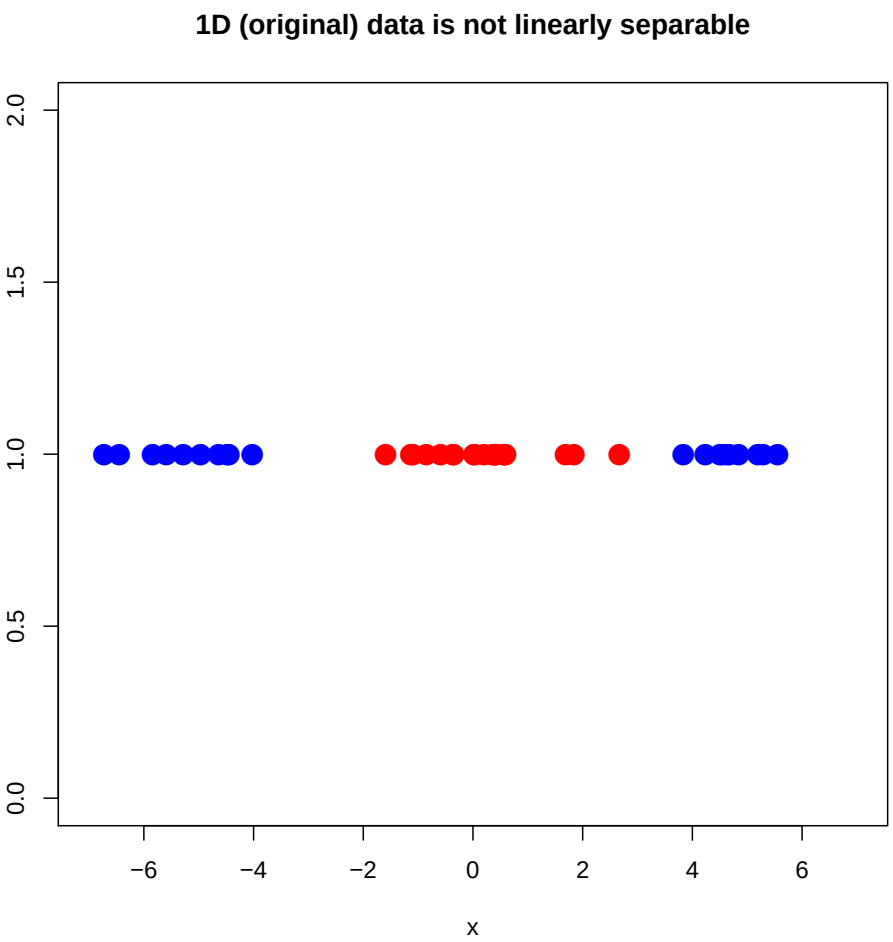
- Map data to a higher dimensional space where classes are linearly separable (artificially increasing number of predictors)
- $(x_1,\ x_2) \rightarrow (1,\ x_1,\ x_2,\ x_1x_2,\ x_1^2,\ x_2^2)$
- Hyperplane in *new* space is a conic section in *original* space

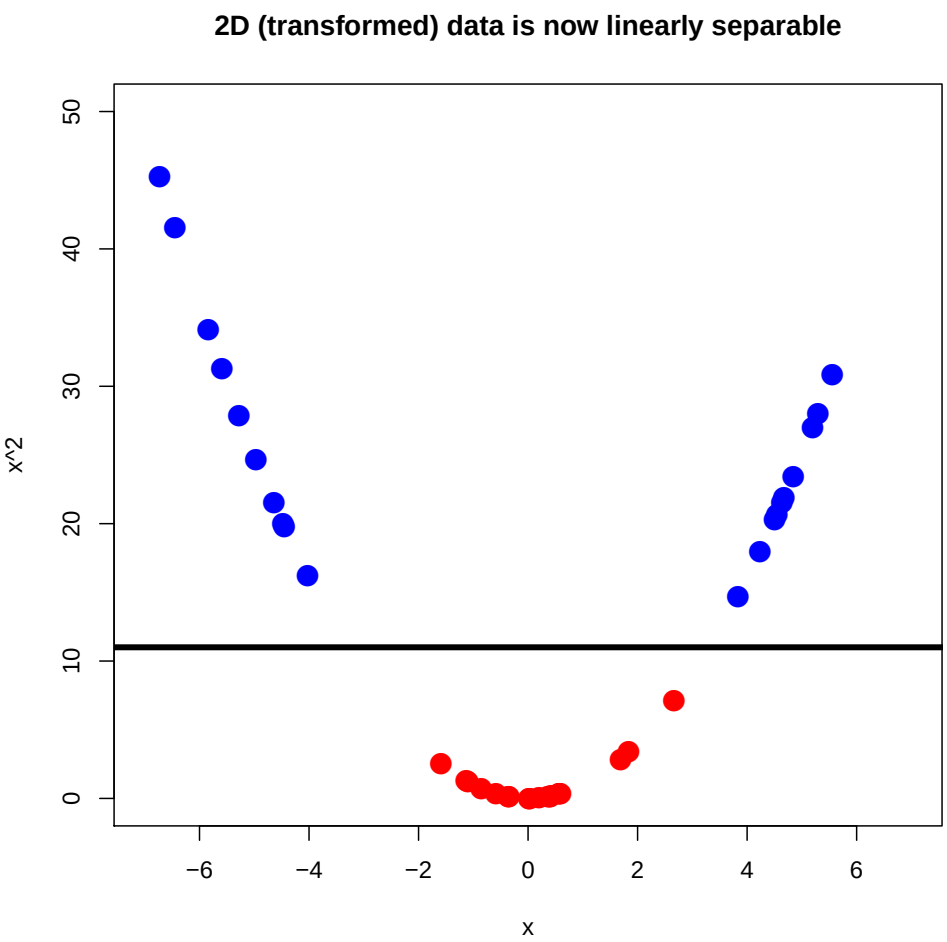


Support vector machines: Simple example from 1D to 2D



Support vector machines: Simple example from 1D to 2D





Support vector machines - The kernel trick

- So our solution is to blow up the dimensions?
- But what about the "curse of dimensionality"?
- Very computationally expensive to work in high dimensions

Support vector machines - The kernel trick

- So our solution is to blow up the dimensions?
- But what about the "curse of dimensionality"?
- Very computationally expensive to work in high dimensions

Kernel trick to the rescue! We work in an *implict* feature space, that is, data is never explicitly computed in higher dimensions. Instead we only need to compute the pair-wise inner product.

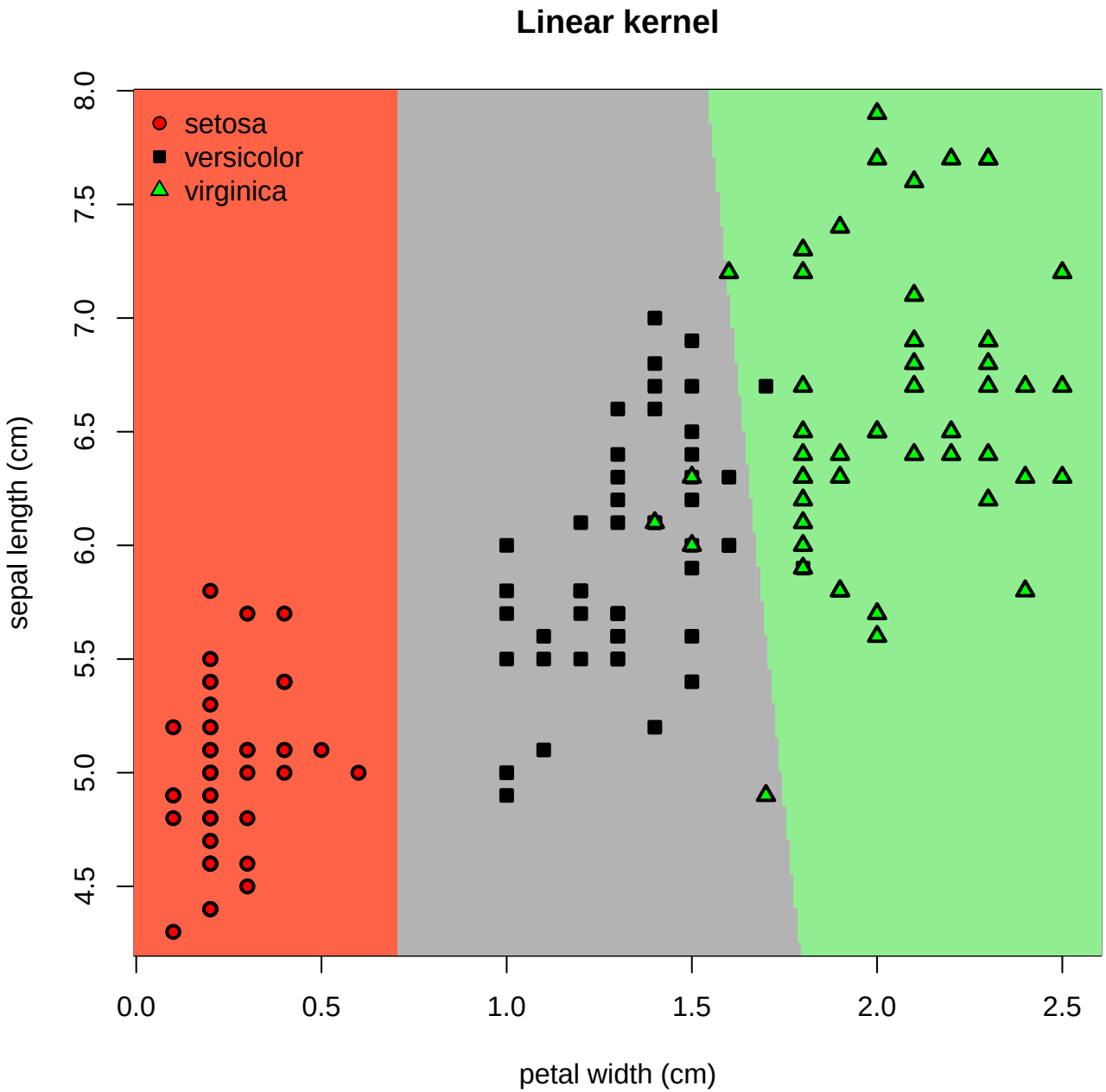
p.s Kernel methods are mathematically intricate and beyond the scope of this workshop so I will stop here

Support vector machines

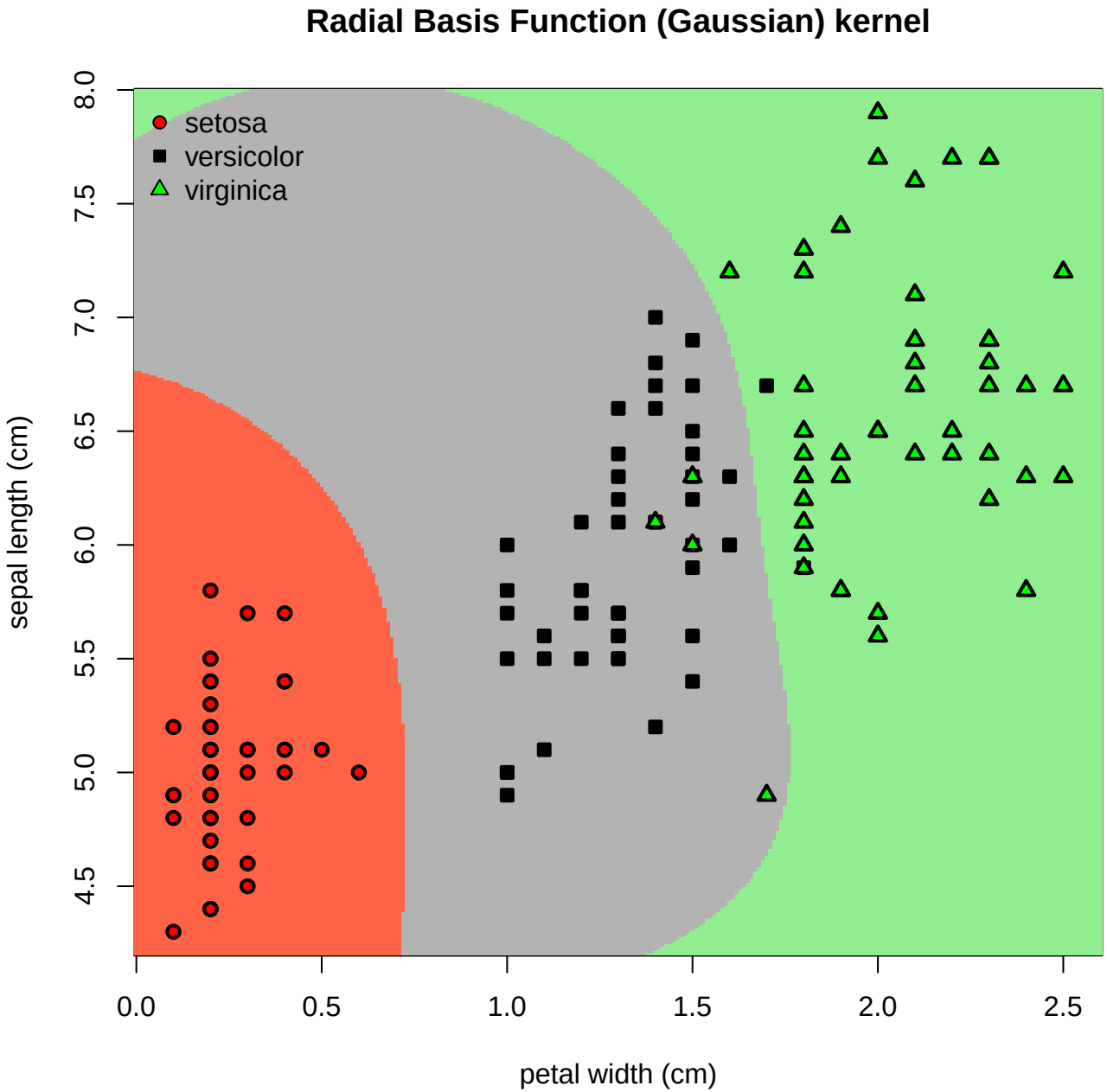
```
library(e1071)
fit <- svm(formula, data, type, kernel)
# formula - an R formula expression e.g y ~ x1+x2
# data - data frame
# type - whether the problem is classification or regression
# kernel - the kernel function e.g "linear" or "radial basis"
```

1. Choose carefully a kernel function
2. Run optimiser to find maximum margin

Support vector machines



Support vector machines



Support vector machines

Pros

- State-of-the-art predictive accuracy
- Less prone to overfitting
- Only need to store the support vectors for the predictive model
- Picking the right kernel gives you flexibility and predictive power
- Global optimum guaranteed

Cons

- Unintuitive/a black box
- Cannot visualise the feature space